Chapter 3
Software for an Intelligent Mathematical Programming System

Matthew J. Saltzman

Abstract Creating and understanding optimization models, instances, and solutions of any significant size present a serious challenge, even to experts in the field. Greenberg pursued an initiative in the 1980s and 1990s to support research and development of computer-assisted technologies to aid decision makers in developing models and investigating model, instance, and solution structures and implications, which he dubbed the Intelligent Mathematical Programming System (IMPS). Among Greenberg's contributions is a suite of software tools that demonstrated the potential for the initiative, including MODLER (a structured model and instance builder), RANDMOD (a structured randomization tool), and ANALYZE (a system for analyzing the structure of model instances and solutions). This paper surveys the capabilities of these tools and their underlying technologies.

3.1 Introduction

It is folk history that in the decades after its early accomplishments in military applications in World War II, operations research (OR) met with mixed success as we discovered both the breadth of applications amenable to OR approaches and the computational hurdles that needed to be overcome. Greenberg wrote in the preface to an unpublished monograph [5], "Due to the explosive growth of inexpensive computer power and to the highly successful applications during the 1960s, *we can solve far larger problems than we can understand.*[1]" However, our aspirations outstripped even those developments. As we can see in retrospect, the "explosive growth" of computer power and algorithm technology of the 1960s was merely the prelude to the dramatic progress that has occurred since.

[1]Emphasis in the original.

M. J. Saltzman (✉)
School of Mathematical and Statistical Sciences, Clemson University, Clemson, SC, USA
e-mail: mjs@clemson.edu

© Springer Nature Switzerland AG 2021

A. Holder (ed.), *Harvey J. Greenberg*, International Series in Operations Research & Management Science 295, https://doi.org/10.1007/978-3-030-56429-2_3

Even for optimization professionals, understanding the intimate relationships between parts of models and the extended impact on solutions of model and instance modifications is utterly impractical without technological assistance. For experts in application areas who have less expertise in optimization, the challenge is even greater. As solver engine power has advanced dramatically over the last few decades, the challenge of understanding the structure of larger and larger models has grown as well.

Computer-assisted analysis tools can be created for individual problems through a collaboration between optimization specialists and problem domain experts, but such tools are expensive: they require repeated, substantial investment of skilled labor. In the late 1970s and early 1980s, Greenberg engaged in a research program "to move some of the art of modeling and analysis to the realm of science [6]."

3.1.1 The Drive for an Intelligent Mathematical Programming System

3.1.2 Technology Context

It is worth recalling the state of computing technology during the period when Greenberg's tools were being developed. IBM announced the introduction of the personal computer (PC) in 1981. Previously, such small computers had been almost exclusively the province of hobbyists, but the PC rapidly began to penetrate the business and academic markets. Those machines offered 16-bit words (32-bit longs), less than a megabyte of RAM, and, typically, 24 × 80-character monochrome screens. Floating-point calculations were either emulated or performed by an extra-cost auxiliary processor. Early hard disk drives offered 5–10 megabytes of space. Color graphics were expensive, and for PCs, offered limited resolution and color depth. The "3M" workstation that was a target of R&D efforts at Carnegie Mellon when the author was in graduate school there included "a megaFLOPS,[2] a megaword,[3] and a megapixel.[4]" Time-sharing mainframes and minicomputers were the main interactive technologies available, but even these had on the order of a few megabytes of memory and tens of megaFLOPS. FORTRAN, COBOL, and to some extent, PL/I were mainframe languages, with FORTRAN 77 the choice for scientific computing. Pascal was the most common structured language on PCs, but it was not standardized, so different compilers supported different features and syntaxes. Unix, C, and C++ were starting to penetrate the small time-sharing system market but were not widely deployed on PCs.

[2]Floating-point operations per second.

[3]RAM.

[4]Monochrome display resolution.

By the 1990s, 32-bit CPUs, a fraction of a gigabyte of RAM, and basic color graphics were the norm for workstations. C and C++ were becoming common languages on PCs and were penetrating the scientific space. Hard disk drives were still tens of megabytes. Linux was just getting its start. Java was gaining a foothold. The World Wide Web, with support for graphics and media, was rapidly becoming the standard for disseminating information on the Internet, which was still mainly the province of government and academia. Remote access to networks was provided over voice lines with modems that could transmit and receive about 10 kb per second.

With regard to mathematical programming, commercial algebraic modeling languages such as GAMS and AMPL existed, but mathematical programming instances were still often created with custom matrix generators. The nearly universal instance interchange format for instances of linear programs (LPs) was IBM's MPS format. MPS was never standardized, so even to this day, different solvers expect slightly different variations. In addition, MPS format provides few mechanisms for expressing special structures. While algebraic modeling packages often used their own file formats for interacting with solvers, these were generally restricted to use with the corresponding packages.

3.1.3 Industrial Sponsorship

Another bit of folk wisdom is that obtaining federal funding for development of software tools to support research in multiple disciplines has historically been more difficult than obtaining funding to carry out traditional "knowledge creation" research. Greenberg also encountered that challenge in the 1980s. In response, he created an industrial consortium to support his IMPS program [6].

Greenberg's consortium proposal was developed in 1984–1985. The first company brought on board was Amoco Oil Co. Later additions included General Research Corporation (apparently now defunct), Shell Research, Ketron Management Science, US West (one of the Baby Bells), and MathPro, Inc. Phase 1 lasted until about 1989. It included development of ANALYZE, MODLER, and RANDMOD and produced over 35 documents, including software manuals and refereed journal articles. Phase 2 was under way when Greenberg's report appeared in 1990, with Phase 3 planned. The author has not located documentation on later phases.

According to Greenberg's self-assessment, there were several features of the consortium model that contributed to its success. To attract consortium members, Greenberg offered clear objectives with associated deliverables, early access to results for consortium members, and inclusion of consortium members in the priority setting process. Several workshops were held in support of the initiative, with presentations of research results and opportunities for collaboration.

3.2 Anatomy and Views of a Model

While mathematicians are comfortable with the algebraic or netform description of an optimization model, subject matter experts may not be comfortable working with those expressions.[5] Greenberg and Murphy [13] provide a taxonomy of LP views, several of which are intended to be more accessible to non-mathematical subject matter experts.

Greenberg and Murphy [13] (and Greenberg in several other publications) describe a mathematical programming model as having the partial structure depicted in Fig. 3.1. They proceed to investigate several different *views* of a mathematical programming model or instance and its solution, each of which may be appropriate for different constituents or may provide a different form of insight into model structure. MODLER and ANALYZE together provide a subset of the views presented in [13]. Commercial algebraic modeling systems generally present only one or two of these collected views.

We denote a (linear) *model* as an abstract representation of a class of *instances*. The instance class is typically infinite and is parameterized by sets of index names or values and numerical parameter and data values. The parameters defining a particular instance class share a common structure, which may be specified with more or less detail to define the structure of the abstract model.

Data objects map to the index sets and the coefficients of the objective, right-hand side, constraint matrix, and bounds. These are the components of a model that change from instance to instance. Sets are considered *symbolic* data, although they can also be described by discrete numerical values. Sets must be discrete because they provide values for indices of discrete objects of other types. For example, sets can index the terms in a summation. (If a set consisted of an interval in $\mathbb{R}$, the summation would decay to an integral, which is outside the scope of these tools.) Data objects can be *explicit* (expressed as a list or table) or *implicit* (expressed as transformations of other sets or tables).

Relations among objects also determine how an instance is generated. *Generation conditions* determine whether decision or data objects appear in a particular instance. *Admissibility conditions* express requirements on data objects such as a numeric range for a table entry or a parameter.

Greenberg and Murphy note that there is some ambiguity regarding what features are considered part of the model and what are part of an instance. In algebraic modeling languages—including MODLER—all objects and relations that are not explicit data must be declared as part of the model definition. Only the specification of explicit set, table, and parameter values distinguish instances.

[5]The term *mathematician* is used somewhat loosely here to refer to someone familiar with the mathematical aspects of optimization models, including their algebraic description, algorithmic solution, and theoretical properties, such as duality relations. *Subject matter experts* are, by contrast, familiar with the terminology related to the application area of a model, but not necessarily with its mathematical properties.

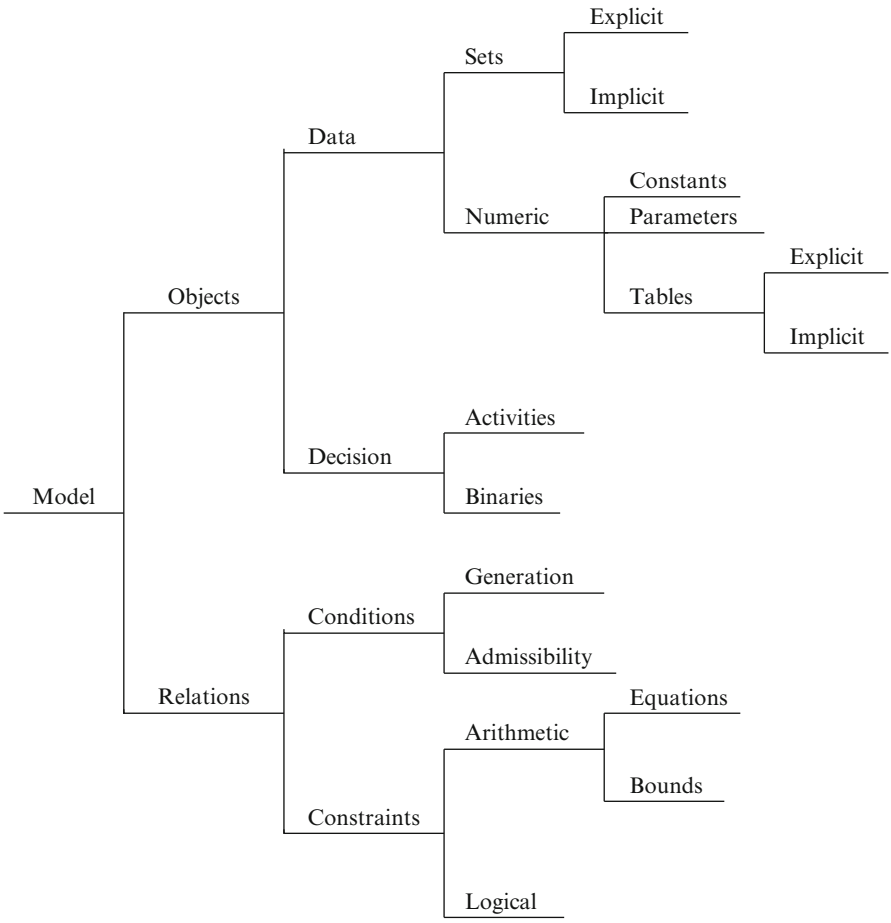


Fig. 3.1 Anatomy of a mathematical programming model

A *view* of a model is a representation of model components in a form that is comprehensible by people involved in the modeling process. These people may have different cognitive expectations for model presentation, and different views can be designed to conform to the expectations of different constituents.

As an example, we map the objects and relations associated with the *capacitated transportation model* to the entities displayed in Fig. 3.1. Model properties include identifying supply and demand points, units of goods available at supply points and required at demand points, and costs and capacities of routes connecting supply and demand points. The explicit sets in an instance definition consist of lists of identifiers for the supply and demand points. Explicit tables provide the number of available units of a good at the supply points, the number of units required at each demand point, and the shipping cost per unit and capacity in units for each

connecting route. Note that the costs and capacities are indexed by an implicit set, namely the Cartesian product of the supply and demand sets. The actual set elements and table values are not specified as part of the abstract model; they must be specified to construct an instance of the model. The decision objects here are activity levels corresponding to the number of units of goods moved from each supply point to each demand point.

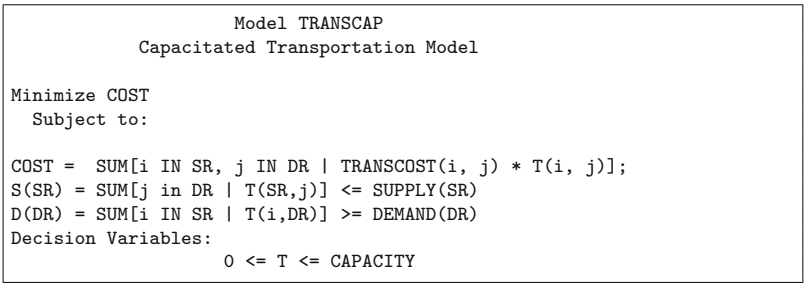
The anatomy diagram does not specify an objective object, but we can define a constraint that specifies the computation of the total shipping cost as the sum over all routes of the unit cost to ship on a route times the amount shipped on that route; then we can specify that quantity is to be minimized. The remaining constraints specify that the amount shipped out of each supply point must not exceed the supply available, that the demand at each demand point must be met, and that the amount shipped on each route must be nonnegative and must not exceed the route capacity.

Greenberg and Murphy provide examples of different views of a capacitated transportation model, and Greenberg [8, 9] provides a collection of models with the MODLER and ANALYZE software with which a user can experiment. Greenberg and Murphy illustrate several views that could be useful to various participants in the modeling process. The views presented in Fig. 3.2 are based on the abstract model of the capacitated transportation example, while those presented in Fig. 3.3 are based on a completely specified instance.

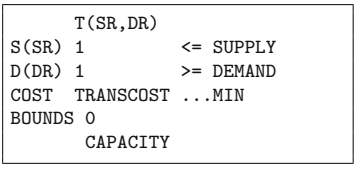
Three of the views in Fig. 3.2 are generated by MODLER. Figure 3.2a presents an algebraic view, Fig. 3.2b is a block schematic view, and Fig. 3.2d is an activity input-output view. A transportation activity in Fig. 3.2d consists of one input (the coefficient from the corresponding supply equation) and one output (the coefficient from the corresponding demand equation). MODLER's views are described in detail in Sect. 3.3. The remaining views in Fig. 3.2 were generated by other tools. Figure 3.2e shows a *netform*, or a network-based model view. The underlying model here is a classical network flow model, so each activity is represented by an arc connecting a supply node at the tail to a demand node at the head. Figure 3.2f shows a condensed version of an *activity-constraint digraph*. Figure 3.2c shows a graphical representation of activity input and output produced by the LPFORM tool [16] (more detail would be available in subordinate screens).

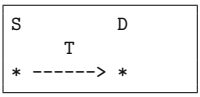
Once a model instance is *instantiated* by assigning values to all data objects, additional views are possible. Figure 3.3 shows some of these views, created by ANALYZE. Figure 3.3a is an algebraic view with coefficients displayed. Figure 3.3b is a block schematic view with coefficient values or ranges included. Figure 3.3d shows a syntax view, where descriptions of objects are expressed in text form using data provided by MODLER. Figure 3.3c displays the sign pattern of entries in the coefficient matrix and rim vectors. Figure 3.3e is an instantiated version of the activity-constraint input/output view in Fig. 3.2d. Figure 3.3f illustrates flows from supply centers to demand centers. ANALYZE's views are described in Sect. 3.5.

Commercial algebraic modeling systems are primarily designed to support an algebraic view, which is familiar to mathematicians but possibly not to other constituents.



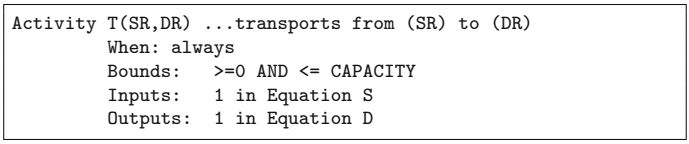
(a)



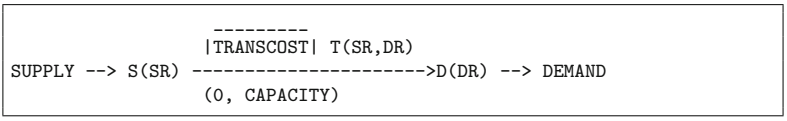


(c)

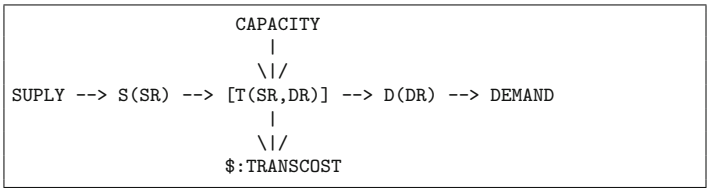
(b)



(d)



(e)



(f)

Fig. 3.2 Some model views of the capacitated transportation problem [13]. (**a**) An algebraic model view. (**b**) A block schematic view. (**c**) A block/link view. (**d**) An activity input/output view. (**e**) A netform view. (**f**) An activity-constraint view (condensed)

```
   MIN COST = TNTNT + TSWSW + 10 TNESW + 10 TSWNE
 50  <= DNE = TNENE + TSWNE
100  <= DSW = TNESE + TSWSW
100  >= SNE = TNENE + TNESE
 50  >= SSW = TSWNE + TSWSW

COL    LO_BOUND   UP_BOUND
----------------------------
TNENE       0          *
TNESW       0         50.000
TSWNE       0         50.000
TSWSW       0          *
```

(a)

```
       T(SR,DR) RHSMMODL
S(SR)  1        <= 50/100
D(DR)  1        >= 50/100
COST   1/10     ...MIN
:LO    0
:UP    50/*
```

(b)

```
       T T T T
       N N S S
       E E W W
       N S N S
       E W E W
COST   + + + + - MIN
DNE    +   +   >  +
DSW      +   + >  +
SNE    + +     <  +
SSW        + + <  +
```

(c)

```
Row syntax has 2 classes
 A row that begins with S limits supply at some supply region.
 A row that begins with D requires demand at some demand region.

Column syntax has 1 class
 A column that begins with T transports from some supply region to
    some demand region.
```

(d)

```
100 --> (SNE) ---> [TNENE] ---> (DNE) --> 50
                     $1
          50 ---> [TNESW] ---> (DWS) --> 100
                     $10
 50 --> (SSW) ---> [TSWNE]  ---> * (DNE)
                     $10
          50 ---> * [TSWNE]
                  [TSWSW] ---> * (DSW)
                     $1
```

(e)

```
      ROW DIGRAPH

Northeast ----> Northwest
          ----> Southwest
Southwest ----> Northeast
          ----> Southwest
```

(f)

Fig. 3.3 Some instance views of the capacitated transportation problem [13]. (**a**) An algebraic instance view. (**b**) A schematic view. (**c**) A sign-pattern view. (**d**) A syntax view. (**e**) An activity-constraint I/O view. (**f**) A flow view (with English translation)

3.3 MODLER: Modeling by Object-Driven Linear Elemental Relations

Custom matrix generators were the primary method of constructing nontrivial instances of linear programming models in the early days of computational optimization. Matrix generators pulled data from whatever sources were necessary and formatted them into MPS-format files for input to solvers. Separate custom programs took solution files in whatever form the solver provided them and produced reports formatted for the decision maker's convenience. As the sizes of instances solvable by computers increased and the reach of personal computers expanded through the 1980s, it became clear that these tools were not adequate to the needs of decision makers who were becoming interested in using optimization in their work. Later in the 1980s, a number of commercial products brought to market the idea of "algebraic modeling languages." AMPL, GAMS, AIMMS, MPL, and LINGO all date from that period.

Algebraic modeling languages share two key capabilities:[6]

- They support the abstract description of a model using an analog of the mathematical notation common in academic writing, with sigma notation for sums and other arithmetic and logical operators. Capabilities include the construction of flexible, abstract indexing sets and multi-subscript parameters, variables, and constraints.
- They separate the specification of the abstract model from the provision of actual values of the indexing set members and the coefficients. Thus, a single abstract description can be reused to specify multiple instances of a problem, simply by providing different data sets to accompany it.

Note that algebraic modeling languages mainly map to a mathematician's view of a problem. The level of abstraction is just what a mathematician thinks about: index sets, coefficients, variables, objective functions, and constraints.

MODLER has a more ambitious agenda [10]. As an interactive system for creating models and instances, MODLER implements an algebraic modeling language. MODLER eschews some of the more esoteric features of commercial algebraic modeling languages and is restricted to formulating linear models; however, it attempts to provide a bridge between the entities and actions that a subject matter expert might consider and the modeling objects (variables, coefficients, constraints, blocks, objectives) that form the mathematician's view. It also supports expression of logical constraints with Boolean variables and automatically converts them to linear inequalities. MODLER's language supports implied indexing and implied summations for expressions with unbound indexing variables.

[6]This definition excludes the simple, row-oriented, written-out expression languages such as LINDO or CPLEX's LP format as well as spreadsheets.

MODLER implements a strict separation of an abstract model from the data associated with an instance. It also supports randomization features that are closely tied to model structure for rapid prototyping of models.

One of MODLER's key features is the ability to generate syntactic data structures for use with ANALYZE. This feature supports expressing results of analyses in natural-language terms that would be familiar to the subject matter expert, as opposed to the language of model formulations that would require a mathematician to interpret. The instance views supported by ANALYZE are described in Sect. 3.5.

MODLER's extensive library of views and queries provides perspective primarily at the level of abstract models.

- The *algebraic view* will be largely familiar to the mathematician. It includes the usual representation of indexed constraints and summations describing a linear program.
- The *block schematic* view is an abstraction of the blocks of variables and constraints that share common names and index sets. The result is a grid with columns corresponding to variable blocks and rows corresponding to constraint blocks. The cells in each row/column indicate where the coefficients are defined. This could be a table, a range of explicit values, etc. Blocks can also appear for logical constraints and bounds.
- The *activity input/output* view shows the model as a collection of transformations. As formalized by Ma et al. [16], transformations represent conversions of form (transforming raw material into product), place (transporting from origin to destination locations), or time (carrying inventory or investments). In a canonical-form LP (minimizing subject to greater-or-equal constraints and nonnegative or bounded variables), an input to an activity is represented by a constraint with a negative coefficient and an output is represented by a constraint with a positive coefficient. MODLER also supports assigning these and other user-defined attributes to sets for display in MODLER and ANALYZE views. MODLER's activity I/O view displays for each activity class a list of constraints where the activity takes an input and where it produces an output.
- *Dependency relations* can be displayed, showing which objects are defined in terms of the sets, parameters, and tables that provide the data for instantiation of an instance of the model. Implicit sets and tables are dependent on the explicit objects that define them, and variables and constraints are dependent on the sets that index them and the parameters and tables that provide their coefficients.

MODLER includes a randomization function that is designed to rapidly prototype instances of a model for testing. Limited randomization can be accomplished interactively from MODLER's console or, more flexibly, from input files that provide explicit set, parameter, and table values. The randomizer can set probabilities for selection among a specified list of ranges or a default range; then random numbers of specified precision are generated with a specified distributions.

3.3.1 Capturing Structure in Instance Representations

MODLER's output is intended to provide input to a solver engine and to the companion tools, RANDMOD (a tool to construct random instances from a template instance) and ANALYZE (MODLER's companion tool for analyzing instances and solutions). The *matrix file* is a standard MPS-format description of the instance, which is input to the solver and to RANDMOD and ANALYZE. The *syntax file* provides a collection of verbal descriptions of objects that can be used with MODLER's description of the model, the matrix file, and the solution report from the solver to display properties of an instance and its solution in natural language.

For generating views and responding to queries regarding instances, Greenberg describes a mapping from object identifiers (variable and constraint group names, index set members, etc.) to instance row and column names in the matrix file. In MPS format, row names, column names, and bound and right-hand side block names are all simple strings of eight characters. (In some MPS extensions, longer names are permitted, but the forms and restrictions are far from universal. These tools generally kept to the most widely supported formats.) ANALYZE and RANDMOD identify substructures and generate views and query responses by matching substrings to patterns. For example, in Greenberg's WOODNET sample model describing production and distribution of lumber, the activity name `TMOSFSE` represents transportation (`T`) of mahogany (`MO`) from a supply point in San Francisco (`SF`) to a demand point in Seattle (`SE`). A syntax for masks supports substring matching to select groups of objects.

3.4 RANDMOD: Controlled Randomization of Linear Programs

RANDMOD [7] is a tool for constructing random instances of linear programs for algorithm testing purposes. Given an input instance specified in an MPS-format matrix file, RANDMOD can produce transformed instances using any of several transformations and generate random values according to any of several distribution classes. The transformations include:

- Augmentation—adding rows to a problem instance constructed from conic combinations of existing inequality rows. The additional rows can be shifted to be strictly redundant or to create degeneracies or infeasibilities.
- Perturbation and scaling—changing row or column bounds or coefficient values.
- Removing bounds.

Row augmentation and perturbation are mutually exclusive operations.

The weights used to construct combinations of rows or to modify coefficient values are randomly generated. The user can specify a range and distribution for a base value, scale factor, offset, and number of modifications for each operation. The

supported distributions are uniform, triangular, normal, and exponential. Transformations can be restricted to submatrices based on name patterns. Each collection of transformations produces a new instance that can be saved in a matrix file with the same naming patterns as the template (except for added rows).

3.5 ANALYZE: A Computer-Assisted Analysis System for Mathematical Programming Models and Solutions

Once an instance of a mathematical programming model is instantiated, a number of views can be produced that present the detailed data provided in context. In addition, if a solution is available, more insights can be provided into the relationships between activities and constraints at that solution. Even if it is determined that no feasible solution exists, it is possible to determine what parts of the model or instance might be responsible for that outcome. ANALYZE can provide all these perspectives and more, and can present them in natural-language form if provided with an appropriate syntax file. In addition, ANALYZE provides a customizable, rule-driven interface for adding new knowledge generation tools for problems with special structure.

A summary of the inputs to ANALYZE and the general classes of outputs are shown in Fig. 3.4. ANALYZE requires at least a *matrix file* describing an instantiated instance, and with only that input, ANALYZE supports a limited set of queries that do not rely on the model's structure. *Dictionaries* and *documents* define the interaction between program and user, mediated by the FLIP subsystem (the FORTRAN Language Interactive Processor), the dialog engine for ANALYZE as well as MODLER and RANDMOD. The *solution file* is the output of any of a handful of solver engines that ANALYZE is able to parse, as there is no widely used format for expressing solutions.

The key to ANALYZE's power as an investigative tool is the *syntax file* provided by MODLER. This file includes the maps from the row and column names in the matrix file to the block structure object names and indices of the original model. It also contains the natural-language descriptions of objects used in ANALYZE's natural-language interface. ANALYZE's reasoning capabilities are driven by rule-based logic. Standard and custom rules are provided via *rule files*. Finally, ANALYZE is capable of interacting with external tools such as Chinneck's IIS (irreducible infeasible subsystem) analyzer [2].

Provided with appropriate inputs, ANALYZE supports sensitivity analysis, various views and queries, model simplification, and interpretation of model and solution structure as well as debugging inquiries such as identifying infeasibilities.

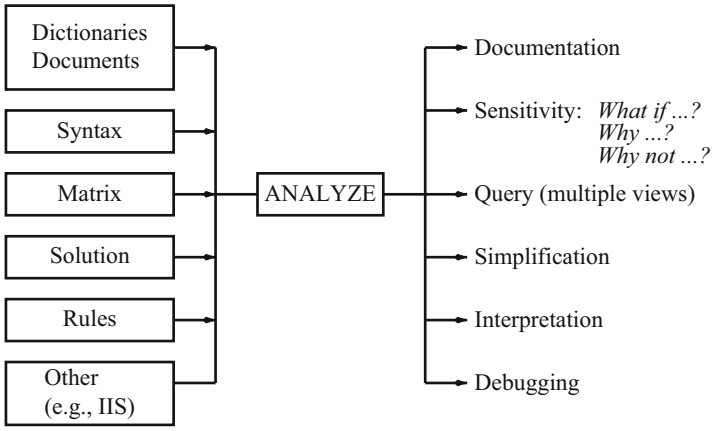


Fig. 3.4 ANALYZE input/output

3.5.1 Views and Analyses

Some available views of an instantiated instance of an LP refine similar views of an abstract model. An algebraic view of a model might involve summations over named index sets of named coefficients, but the instance view can show the actual index and coefficient values. The block schema for a model shows groupings of rows and columns by name, but the instance view can show ranges of coefficients. See related views in Figs. 3.2 and 3.3 for comparison. ANALYZE can also display constraint matrix sign patterns. While the lack of graphics capabilities limited the size of such displays, they were still useful for selected submatrices.

In addition to refinements of model views, there are many ways to explore relationships among components of instances and solution values. By tracing through submatrices associated with active resource constraints and basic activities, ANALYZE can provide information about the makeup of shadow prices and reduced costs, marginal substitution rates, sensitivity of the solution to changes in coefficient values, and other properties of the instance and solution. Those insights can be presented through displays of objects and their properties in diagrams or tables. By using the natural-language descriptions of objects in the syntax file, ANALYZE can also represent its findings in verbal summaries.

The *fundamental digraph* of an instance [4] is a directed bipartite graph with a node for each row and column and an arc connecting row i to column j if matrix coefficient a_{ij} is negative and an arc connecting column j to row i if a_{ij} is positive. For an LP in the canonical form (minimize subject to greater-or-equal constraints and nonnegative variables), one can interpret a negative coefficient as indicating that the row resource is an input to the column activity. A positive coefficient indicates that the row resource is an output of the column activity.

The fundamental digraph can be projected onto the row or column node sets, with an arc between rows in the former or columns in the latter corresponding to directed paths of length 2 in the fundamental digraph. The row digraph captures transformations between resources connected by an activity that takes one resource (the tail) as input and produces another (the head) as output. The column digraph captures precedence, in which one column (the tail) produces a resource that another activity (the head) consumes. ANALYZE can display subgraphs of these graphs to visualize these relations.

While the general question of whether a constraint is redundant has the same complexity as solving the original LP, some redundancies can be verified through the same sorts of analyses as those listed above. ANALYZE can also diagnose infeasibilities using a successive bounding procedure or by hooking to an external engine that implements Chinneck's IIS detector.

3.5.2 Algorithmic Analysis

ANALYZE includes several algorithms and heuristics that support a deeper understanding of the interactions between model instances and solutions than is afforded by simply looking at activity levels and dual prices. The key algorithms in ANALYZE's repertoire include:

- *Path tracing* builds a submatrix that includes rows corresponding to the resources associated with an activity or subset of activities and all the activities that interact with the activity of interest. From that submatrix, ANALYZE can determine the impact of marginal changes in the activity of interest.
- *Basis rearrangement* permutes basis rows and columns to bring the basis matrix to a triangular or near-triangular form.
- *Rates of substitution* can be computed by completing the product-form factorization of the triangularized basis and invoking the FTRAN and BTRAN procedures from the simplex method (to solve $B\boldsymbol{x} = \boldsymbol{a}$ and $B^T\boldsymbol{\pi} = \boldsymbol{c}$, respectively, where $\boldsymbol{a}$ is a column of the constraint matrix and $\boldsymbol{c}$ is a subvector of the objective).
- Some cases of *redundancy* can be detected by computing ranges on basic variables that maintain feasibility as nonbasic variables are set to their most permissive bounds. If the upper or lower bounds on the left-hand sides are tighter than the upper or lower bounds on the right-hand side, the corresponding constraint is redundant.
- *Primal and dual bounds* can be reduced sequentially until infeasibility is detected or the bound reduction process stabilizes.
- *Logical implications* for binary variables can be imputed based on constraint left-hand side bounds.

3.5.3 *The Rule Base*

Rule-based reasoning is one research thrust of artificial intelligence. The idea is to capture the thought process of an expert analyst in the field of interest in a form that can be carried out automatically by a computer. ANALYZE contains a rule-based reasoning engine that includes a number of standard analytical procedures such as interpreting a shadow price or identifying an embedded network. The rulebase is extensible and customizable so that new analyses for special problem structures can be implemented.

Rules can be invoked by the user and can in turn implement algorithms automating the steps of an analysis, such as identifying the contributions of activities to a shadow price or the contribution of resources to a reduced cost in a problem instance. Rules can invoke the core algorithms described in Sect. 3.5.2, where the components that contribute to an interpretation may depend on special structure of the problem.

3.6 WRIP: A Workbench for Research in (Linear) Programming

In 1991, Greenberg and Marsten released a package [12] containing the three analysis tools described here together with an LP solver: Marsten et al.'s OB1 [1, 15, 17, 18]. OB1 is a FORTRAN code that includes Marsten's XMP simplex solver and several different interior-point solvers, plus a crossover code to recover a basic optimal solution from an optimal interior-point solution. The package also includes test instances from Netlib and elsewhere [3, 14] as well as tools for visualization.

Greenberg and Marsten's view of a workflow for experimenting is pictured in Fig. 3.5. A matrix file for a base LP instance could be selected from a library or created using MODLER. The LP could be solved with OB1 or processed through RANDMOD to create additional, similar instances. Solutions from OB1 could be analyzed with ANALYZE, and the output of RANDMOD, OB1, and ANALYZE could be fed to a statistical analysis of, for example, solver performance. The results of the analysis could be reported and could be fed to RANDMOD to produce additional instances for further testing.

3.7 Conclusion

The Intelligent Mathematical Programming System initiative spearheaded by Greenberg in the 1980s and 1990s was an ambitious program to harness emerging computing power to enhance the analyst's ability to formulate, analyze, and reason about optimization problems in the context of decision support systems. Greenberg's

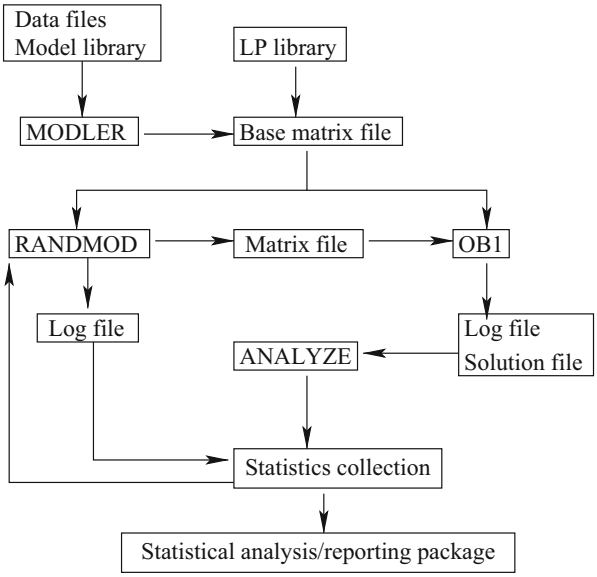


Fig. 3.5 Job flow for experimental analysis

1996 bibliography [11] lists over 500 references, which are classified as relevant to background, analysis, discourse, formulation, model management, and software engineering and implementations, plus relevant general knowledge. Greenberg himself is listed as author or coauthor on nearly 50 of the publications. But among his most influential contributions to the initiative is the fact that he put into practice the principles that he developed and assembled by publishing the software packages MODLER, RANDMOD, and ANALYZE.

While some of the knowledge developed through the initiative and related efforts has been integrated into widely used tools, many of the capabilities of Greenberg's codes have not been so widely deployed. Compiled versions of MODLER and ANALYZE for Microsoft Windows are distributed with user's guides currently available from Springer [8, 9]. Windows and Linux executables for MODLER and ANALYZE and Windows executables for RANDMOD were available for download from Greenberg's University of Colorado at Denver Web pages. These can still be run on systems available as of this writing. Source code for ANALYZE exists and should eventually be available as open source, once proper permissions can be secured. Sadly, source for MODLER and RANDMOD appears to be lost to history, unless some kind reader has an archive that they can share with this author.

References

1. I. Adler, N. Karmarkar, M.G.C. Resende, G. Viega, Data structures and programming techniques for the implementation of Karmarkar's algorithm. ORSA J. Comput. **1**(2), 84–106 (1989)
2. J.W. Chinneck, E.W. Dravnieks, Locating minimal infeasible constraint sets in linear programs. ORSA J. Comput. **3**, 157–168 (1991)
3. D.M. Gay, Electronic mail distribution of linear programming test problems. Math. Program. Soc. Committee Algorithms (COAL) Newslett. (1985)
4. H.J. Greenberg, A new approach to analyze information contained in a model, in *Energy Models Validation and Assessment*, ed. by S.I. Gass, vol. 569 (National Bureau of Standards, Gaithersburg, 1978), pp. 517–524
5. H.J. Greenberg, Foundations for an intelligent mathematical programming system. Draft monograph (1988)
6. H.J. Greenberg, An industrial consortium to sponsor the development of an intelligent mathematical programming system. Interfaces **20**(6), 88–93 (1990)
7. H.J. Greenberg, RANDMOD: a system for randomizing modifications to an instance of a linear program. ORSA J. Comput. **3**(2), 173–175 (1991)
8. H.J. Greenberg, *A Computer-Assisted Analysis System for Mathematical Programming Models and Solutions: A User's Guide for* ANALYZE. Operations Research/Computer Science Interface Series, vol. 1 (Springer, Berlin, 1992)
9. H.J. Greenberg, *Modeling by Object-Driven Linear Elemental Relations: A User's Guide for* MODLER. Operations Research/Computer Science Interface Series, vol. 2 (Springer, Berlin, 1992)
10. H.J. Greenberg, MODLER: modeling by object-driven linear elemental relations. Ann. Oper. Res. **38**, 239–280 (1992)
11. H.J. Greenberg, A bibliography for the development of an intelligent mathematical programming system. Ann. Oper. Res. **65**, 55–90 (1996)
12. H.J. Greenberg, R.E. Marsten, WRIP: a workbench for research in (linear) programming. Software Manual (1991)
13. H.J. Greenberg, F.H. Murphy, Views of mathematical programming models and their instances. Decis. Support Syst. **13**, 3–34 (1995)
14. I.J. Lustig, An analysis of an available set of linear programming test problems. Comput. Oper. Res. **16**(2), 173–184 (1989)
15. I.J. Lustig, R.E. Marsten, D.F. Shanno, On implementing Mehrotra's predictor-corrector interior-point method for linear programming. SIAM J. Optim. **2**(4), 435–449 (1992)
16. P.-C. Ma, F.H. Murphy, E.A. Stohr, A graphics interface for linear programming. Commun. ACM **32**(8), 996–1012 (1989)
17. R.E. Marsten, M.J. Saltzman, D.F. Shanno, G.S. Pierce, J.F. Ballintijn, Implementation of a dual affine interior point algorithm for linear programming. ORSA J. Comput. **1**(4), 287–297 (1989)
18. R.E. Marsten, R. Subramanian, M. Saltzman, I. Lustig, D. Shanno, Interior point methods for linear programming: just call Newton, Lagrange, and Fiacco and McCormick! Interfaces **20**(4), 105–116 (1990)